《Bitcoin Developer Guide》中文版

0dayZh



目錄

比特币开发者指南 | Bitcoin Developer Guide

当前版本 | Current Version

Name	Value
source	https://github.com/bitcoin-dot-org/bitcoin.org
commit	000887a2f4b67c2345ea1c91578cee44c0f90088

关于译本 | About Translation

该译本的所有内容被托管在 GitHub 上，你可以通过 GitBook 进行在线阅读或下载 PDF、EPUB、MOBI 的格式。

如果你发现有任何的翻译出入或者文字书写错误，欢迎 fork 提 pull request，也可以提 issues 提出对译本的建议及意见。

捐献 | Donation

如果你觉得该译本对你有所帮助并希望为我做些事情作为回馈，除了提出修改意见外，还可以捐赠比特币或者购买亚马逊心愿清单中的物品。

- Bitcoin: 1EdMCu32fzGo5iNCyQuMZhHboSe2gzqQPg
- Amazon 心愿单

进度 | Progress (1 / 8)

- [x] 区块链
  - [x] 区块链总览
  - [x] 工作量证明
  - [x] 区块高度及分叉
  - [x] 交易数据
  - [x] 一致性规则变更
  - [x] 发现分叉
- [x] 交易
  - [x] P2PKH 脚本验证

    - [x] P2SH 脚本
    - [x] 标准交易
    - [x] 非标准交易
    - [x] 签名哈希的类型
    - [x] Locktime And Sequence Number
    - [x] 交易手续费及变更
    - [x] 避免 公钥和私钥重用
    - [x] 交易的延展性
- [ ] 合约
    - [x] 托管和仲裁
    - [x] 微支付通道
    - [ ] CoinJoin
- [ ] 钱包
    - [x] 钱包程序
    - [x] 完整服务的钱包
    - [x] 只用于签名的钱包
    - [x] 离线钱包
    - [x] 硬件钱包
    - [x] 只用于发布的钱包
    - [ ] 钱包文件
    - [ ] 私钥格式
    - [ ] 钱包导入格式
    - [ ] Mini 私钥格式
    - [ ] 公钥格式
    - [ ] 分级确定性密钥生成
    - [ ] Hardened Keys
    - [ ] 存储根种子
    - [ ] Loose-Key 钱包
- [ ] 支付流程
- [ ] 运作模式
- [ ] 点对点网络
- [ ] 挖矿

简介 | Introduction

此开发者指南旨在提供给你理解比特币的信息，让你能够开始构建以比特币为基础的应用，并不是一本规范手册。为了尽可能的理解该文档，你可能需要安装最新版本的 Bitcoin Core，你可以通过源码或者预先编译好的可执行文件来获取到。

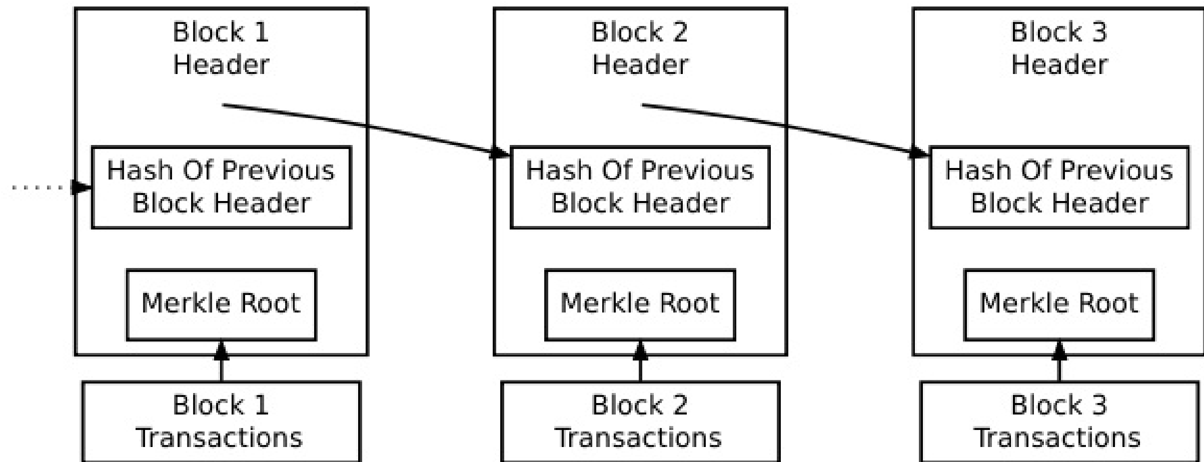
关于比特币开发的问题最好在比特币社区中提出。对于在 Bitcoin.org 上的文档相关的错误或者意见可以通过提交 issue 或者发送邮件到比特币文档邮件列表进行。

区块链 | Block Chain

区块链是所有交易保持有序及时序的一个记录，它为比特币提供了公开的账本。这套系统被用于防止双花和对之前交易的修改。

在比特币网络中的每个完全节点存储着一个区块链，只包含被该节点验证过的区块。当多个节点在其区块链中都有了同样的区块时，他们被认为是一致的（consensus）。这些节点用来维持一致性的验证规则被称作一致性规则（consensus rules）。本节会涉及很多关于一致性规则的描述，而其被 Bitcoin Core 所使用。

区块链总览 | Block Chain Overview

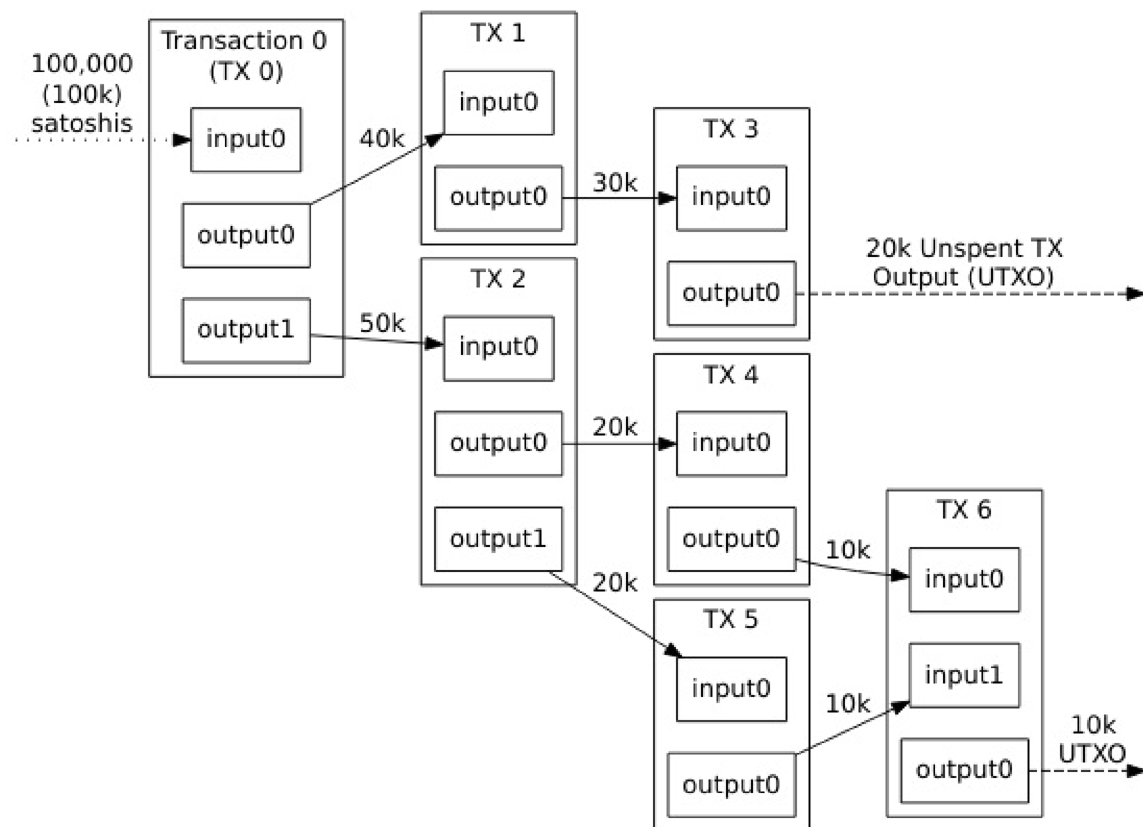


Simplified Bitcoin Block Chain

上图展现了一个简化版的区块链。一个或多个新的交易被收集到一起成为了一个区块（block）的交易数据部分。每个交易的副本都会被哈希，然后这些哈希被配对，被哈希，再配对，再哈希，直到只有一个哈希值，这就是 merkle 树的 merkle 根节点。

merkle 根节点存储在区块头。每个区块同时存储着上一个区块头的哈希值，从而将区块连接成链。这样确保了一个交易在没有对纪录其的区块及所有之后区块更改前是不可被更改的。

交易也是被链在一起的。比特币钱包给人的印象是聪（satoshis）从钱包发出再到钱包，然而比特币实际上是从一个交易到另一个交易。每笔交易花费着从之前的一笔或多笔交易中收到的聪，所以一笔交易的输入是上一笔交易的输出。



Triple-Entry Bookkeeping (Transaction-To-Transaction Payments) As Used By Bitcoin

一笔交易可以有多个输出，就好像同时发送给多个地址一样，但是在区块链中一笔交易的每个输出只能被当作输入使用一次。任何后续的引用都是被禁止的双花（一种将同一笔聪花费两次的尝试）。

输出与交易 id（TXIDs）绑定在一起，交易 id 是签名的交易的哈希值。

因为一笔交易的每个输出只能被花费一次，所以在区块链中的所有交易的输出可以被分类为已花费交易输出和未花费交易输出（UTXOs）。为了让支付有效，必须使用 UTXOs 作为输入。

除了 coinbase 交易（稍后描述）外，如果一笔交易的输出值超过了输入值，这笔交易将被拒绝，但是如果输入值大于输出值，那么其间的差值将会被创建包含该笔交易区块的比特币挖矿工认定为交易手续费。比如，在上图中，每笔交易的花费都比其输入的总和少 10,000 聪，显而易见的支付了 10,000 聪的交易手续费。

工作量证明 | Proof Of Work

区块链是通过网络上匿名的节点协同维护的，所以比特币需要每个区块证明在创建他们的过程中投入了足够多的工作量，从而确保那些想篡改历史区块信息的不可信懒的节点必须要比只想添加新区块到区块链的诚实节点付出更多的工作量。

将区块链在一起使得只修改一个交易而不修改所有后续交易成为不可能。因此随着新的区块被添加到区块链，修改一个特定区块的成本在不断提升，从而放大了工作量证明的作用。

比特币中的工作量证明利用了密码学哈希的伪随机性。一个好的密码学哈希算法可以将任意数据转换为看似随机的数字。如果原始数据的任何地方被更改了，然后重新做哈希，一个新的看似随机的数字就产生出来了，所以通过更改原始数据预测哈希值是不可能做到的。

为了证明你为创造一个区块做了一些工作，你必须算出一个不超过某个特定值的区块头的哈希值。比如，如果这个哈希值最大为 $2^{256}-1$，你可以证明你算出一个小于 2^{255} 的哈希值做了至多两次组合。

在上面给出的例子中，你可能在某次尝试中就得到了一个成功的哈希值。你甚至可以估算出算出低于目标阈值的概率。比特币设定了一个线性的概率，目标阈值越小，你需要尝试计算的哈希值次数越多（平均来说）。

当一个新的区块的哈希值至少与一致性协议期望的难度值相当时，它才会被加到区块链中。每产生 2,016 个区块，比特币网络使用存储在区块头中的时间戳来计算这 2,016 个区块的第一个和最后一个区块的时间差值，理想的值为 1,209,600 秒（及两周）。

- 如果生成这 2,016 个区块所用的时间比两周短，那么预期的难度将会相应的增长（大概为 300%），这样在哈希算力保持不变的情况下，之后的 2,016 个区块的生成应该刚好会用两周的时间。
- 如果生成这些区块的时间超过了两周，出于同样的原因，预期的难度也会相应的递减（大概为 75%）。

（注意：在比特币内核实现中存在一个差一错误，导致算力难度的是由 2,016 个区块中的 2,015 个区块的时间戳来更新的，产生了一个微小的偏差。）

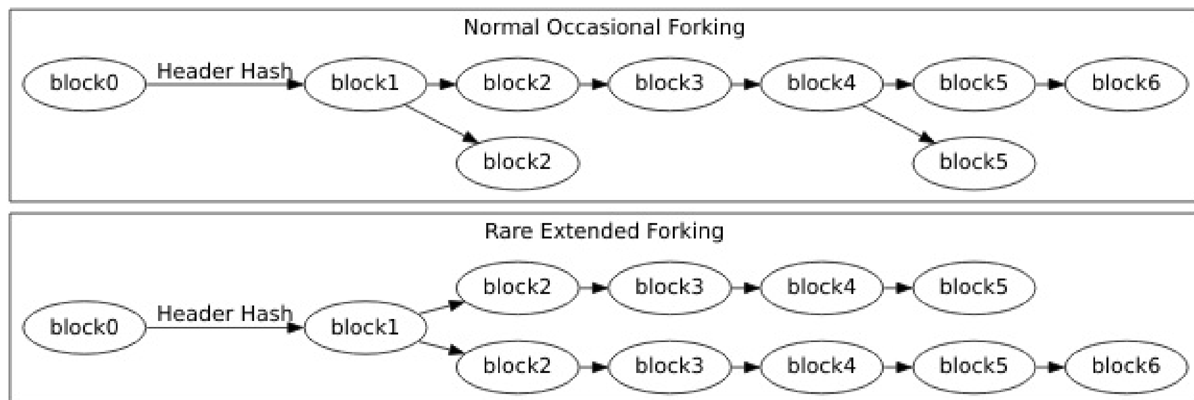
因为每一个区块头的哈希值必须是比目标阈值要小的，并且每个区块与先前的一个链接在一起，所以想对一个区块做更改需要（一般来说）付出整个比特币网络从原始区块时间点到当前时间点所消耗的哈希算力。只有当你拥有了整个网络大部分的哈希算力时你才有可能发起一个有效的 51 攻击来篡改交易纪录（当然，需要指出的是，即便你拥有的算力小于 50%，仍然有机会发起这样的攻击。）

区块头提供了一些便于修改的字段，例如 nonce 字段，因此获取新的哈希值时并不需要为新的交易而等待（译者注：我理解这里的意思是通过缓存状态而避免每次计算哈希都需要遍历交易记录。）同时，只有区块头的 80 个字节被用来当作工作量证明做哈希，所以并不会因为

区块读取其包含的大量的交易数据所产生的 I/O 延时而延缓哈希运算，并且在添加一个额外的交易数据时只需要重新计算一下 merkle 树的原始节点哈希即可。

区块高度及分叉 | Block Height And Forking

任何成功计算出那个比目标阈值低的区块头哈希值的矿工，都可以将这整个区块加到区块链中（假设这个区块是有效的）。这些区块通常是通过区块高度来定位的，所谓区块高度是指某个区块与比特币第一个区块（及区块 0，众所周知的创世区块）之间的区块个数。比如，区块 2016 就是第一次难度调整的位置。



多个区块可以具有同样的区块高度，这在两个或更多的矿工同时创建一个区块时是很常见的。这导致了如上图所示的明显的分叉。

最终，矿工会产出一个新的区块，它附加在且仅附加在同时被挖出的区块中的一个之后。这样使得某一个分叉比其他分叉更为健壮。假设一个分叉只包括有效的区块，普通的节点总是跟随更长的链，而放弃陈旧的短的分叉。（陈旧的区块常常被叫做孤链，但是该术语也被用于描述确实没有父区块的区块。）

如果一些矿工有其他的意图，长期的分叉是有可能出现的，比如一些矿工在延续区块链工作，而同时另一些矿工试图通过发起 51% 攻击篡改交易纪录。

因为多个区块可以在区块链分叉时具有同样的区块高度，所以区块高度不应该用来做全局的唯一标示符。通常，区块使用区块头的哈希值来做标示（大多数时候是字节倒序的十六进制表示）。

交易数据 | Transaction Data

每一个区块必须包含一笔或者多笔交易。这些交易的第一笔必须是一个 coinbase 交易，也被称作 generation 交易，它包含了这个区块所有花费和奖励（由一个区块的补贴和这个区块任何交易产生的交易费用构成）。

一个 coinbase 交易的 UTXO 有一个特殊的条件，那就是在之后的 100 个区块内都不能被花费（被用作输入）。这临时性的杜绝了一个矿工花费掉因为分叉而可能被判定为陈旧区块（这个 coinbase 交易因此被销毁掉）中所包含的补贴和交易费。

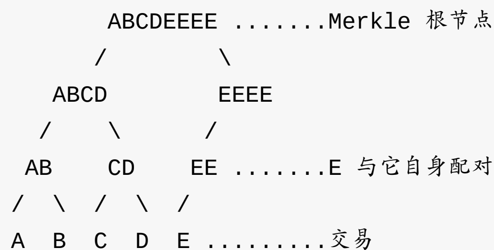
区块并不要求一定包含非 coinbase 的交易，但是矿工们为了把他们的交易手续费包含其中总是会带上额外的交易。

所有的交易，包括 coinbase 交易，都会被编码为二进制的 rawtransaction 格式存入区块。

通过对 rawtransaction 格式做哈希得到了交易标识符（txid）。从这些 txids 中，通过将一个 txid 与另一个 txid 配对然后做哈希运算，最终构建了 merkle 树。如果 txids 的个数为奇数，那么没有被配对的那个 txid 将会与他自身的一个副本配对做哈希。

以上得到的哈希值再一一配对，让一个哈希值与另一个再做哈希运算。任何没有可配对的哈希值与自身配对做哈希。这个过程迭代进行直到只剩下一个哈希值，这就是 merkle 根节点。

例如，如果交易只是连接（没有做哈希运算）在一起，那么具有 5 个交易的 merkle 树应该如下图所示：



```
         ABCDEEEE .......Merkle 根节点
        /        \
     ABCD         EEEE
    /    \       /
   AB     CD    EE .......E 与它自身配对
  /  \   /  \  /
  A   B  C   D E .........交易
```

在简化支付验证（SPV）提案中指出，merkle 树允许客户端通过一个完整节点从一个区块头中获取其 merkle 根节点和中间哈希值列表来验证一个交易被包含在这个区块中。这个完整节点并不需要是可信的，因为伪造区块头十分困难而中间哈希值是不可能被伪造的，否则验证就会失败。

例如，要验证交易 D 被加到了区块中，一个 SPV 客户端只需要一份 C，AB 和 EEEE 的副本进而做哈希运算得到 merkle 根节点，客户端并不需要知道任何其他交易的信息。如果这 5 个交易的大小都是一个区块的最大上限，那么下载整个区块需要下载 500,000 个字节，但下载树的哈希值和区块头仅仅需要 140 个字节。

注意：如果在同一个区块中找到了相同的 txids，那么 merkle 树可能会出现与另一个区块的碰撞，归因于 merkle 树对非平衡的实现（对孤立的哈希值做复制）会将一个区块中一些或全部的重复内容移除掉。从对于单独的交易具有相同的 txid 是不现实的角度来看，merkle 树的实现不会对诚实的软件增加负担，但如果一个区块的无效状态要被缓存那么就必须做检查，否则，一个移除了重复交易的有效区块会因为具有相同的 merkle 根节点和区块哈希而被缓存的无效状态拒绝，导致了编号为 CVE-2012-2459 的安全问题。

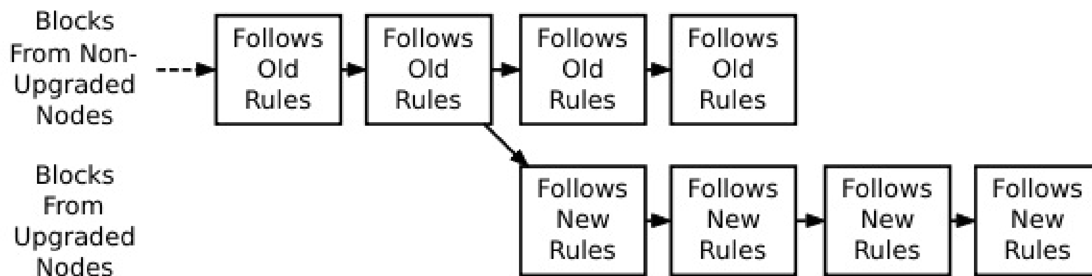
一致性规则变更 | Consensus Rule Changes

为了维持一致性，所有的完整节点使用相同的一致性规则验证区块。然后，有时为了加入新特性或者防治网络滥用一致性规则会被更改。当新的规则被实施后，会在一段时间内出现已更新节点遵循新的规则而未更新的节点遵循旧的规则，这导致两种可能的一致性分歧的出现：

1. 一个区块遵循新的一致性规则，它将被已更新节点接受而被未更新节点拒绝。例如，一个新的交易特性被用于区块内部，那么已更新节点便可以理解这一特性并接受它，但是未更新节点则因为它违背了旧的规则而拒绝了它。

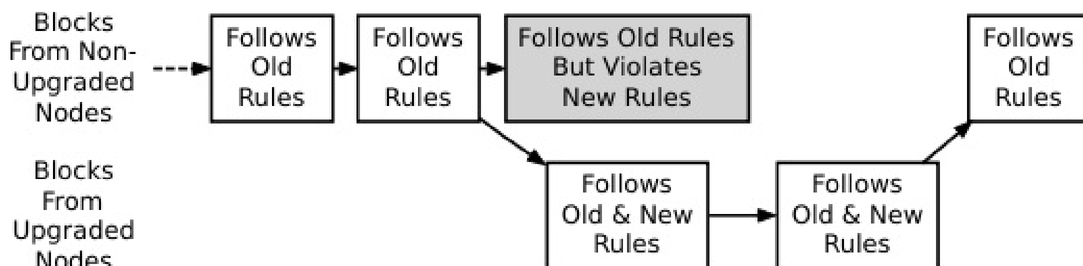
2. 一个区块违反了新的一致性规则而被已更新节点拒绝，但却被未更新节点接受。例如，一个不合理的交易特性被用在一个区块内，已更新的节点因为新规则拒绝了它，但未更新的节点遵循旧的规则所以接受了它。

在第一种情况中，即未更新节点拒绝的情况，从未更新节点获取到区块链信息的挖矿软件会拒绝从已更新节点获取数据的挖矿软件在同一条链条上构建区块。这样导致了永久性的分叉链，一条是已更新节点的，一条是未更新节点的，这被称为硬分叉。



A Hard Fork: Non-Upgraded Nodes Reject The New Rules, Diverging The Chain

在第二种情况中，即已更新节点拒绝的情况，如果已更新节点掌握大部分的算力就有可能避免永久性分叉。在这种情况下，因为未更新节点会和已更新节点接受相同的区块而使已更新节点构建了更长的链，这样未更新节点便会接受更长的有效区块链。这被称作软分叉。



A Soft Fork: Blocks Violating New Rules Are Made Stale By The Upgraded Mining Majority

尽管一个分叉在区块链中是一个实实在在的分歧，但是对一致性规则的更改被经常描述为有可能出现软分叉或者硬分叉。比如，“扩展区块大小上限到 1 MB 需要一个硬分叉。”在这个例子中，一个区块链的硬分叉并不是一定需要，但是他却是一种可能的结果。

资源：BIP16，BIP30 和 BIP34 的实现被当作可能导致软分叉的变更。BIP50 描述了一种意外的硬分叉（通过暂且对已更新节点降级来化解）和一种当暂且的降低被移除后的有意的硬分叉。由 Gavin Andresen 写的一篇文档描绘未来的规则更改该如何实现。

发现分叉 | Detecting Forks

在出现上面提到的两种分叉中，未更新的节点都可能使用和发布不正确的信息，导致一些会出现经济损失的情况。特别的，未更新的节点会传播并接受那些被已更新节点认定为无效的交易，这些交易将永远不会成为全网最长区块链的一部分。未更新节点也可能拒绝传播已被或将要被加到最佳区块链的区块和交易，这样它们提供的就是不完整的信息。

Bitcoin Core 包含了通过监控区块链工作量证明而发现硬分叉的代码。如果一个未更新的节点接收到的区块链头证明至少有六个区块比这个节点认定的最佳区块链有更多的工作量证明，这个节点就会在 `getinfo` RPC 命令结果中报错并且在开启了 `-alertnotify` 时运行 `-alertnotify` 。这提醒运营人员这个未更新节点无法切换到那个本应该是最佳的区块链上去。

完整的节点同时会检查区块和交易的版本好。如果最近的几个区块或者交易的版本比节点使用的要高，这可以假定为该节点没有在使用当前的一致性规则。Bitcoin Core 0.10.0 通过 `getinfo` RPC 命令报告这一情况，并且如果 `-alertnotify` 被设置了会运行 `-alertnotify` 。

在以上的情况中，明显地来自一个未使用当前一致性规则节点的区块和交易数据是一定不应该被信赖的。

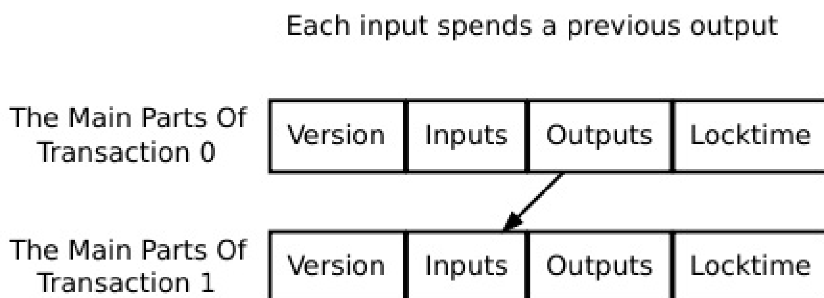
连接到完全节点的 SPV 客户端可以通过连接多个完全节点确保他们在同一区块高度来判定是否出现了硬分叉，要加上或减去一些账户的区块因为存在交易的延时和陈旧的块。如果出现了分叉，客户端可以与具有短链的节点断开连接。

SPV 客户端最好也对区块和数据的版本号增长做监控，从而确保它们在使用同样的一致性规则处理接受交易和创建新的交易。

交易 | Transactions

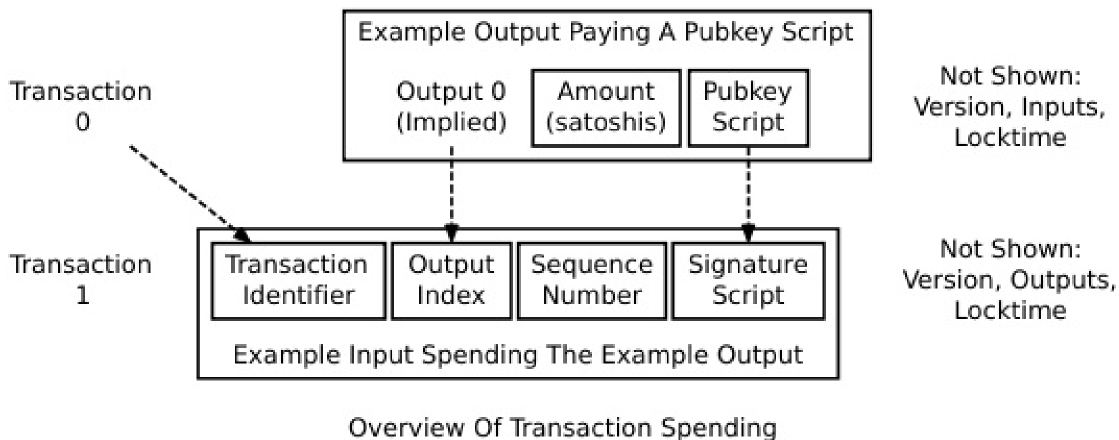
交易让用户可以花费聪（satoshis）。每一笔交易是由多个部分构成的，使单纯直接的支付和复杂的交易成为可能。这一章节将对其每一部分做详解并展示如果将它们拼接起来构建一个完成的交易。

为了便于理解，这一节假设不存在 coinbase 交易。只有比特币矿工可以创建 coinbase 交易，它们对于下面会提到的很多规则都将是例外。这里并不会指出 coinbase 对哪一条规则是例外，你可以通过阅读本指南中区块链章节中的 coinbase 交易小节来做了解。



上图展示了一个比特币交易的主要部分。每一个交易至少有一个输入和一个输出。每一个输入会花费掉上一个输出的聪。一个输出会作为一个未花费交易输出（UTXO）等待着作为输入被花费掉。当你的比特币钱包显示你有 10,000 聪余额时，它眞实的表意是你有 10,000 聪作为 UTXOs 在等待着。

每一个交易具有一个四字节的交易版本号，它告知比特币节点和矿工应使用哪一套规则来验证它。这使得开发者在为未来的交易创建新规则时可以不验证之前的交易。

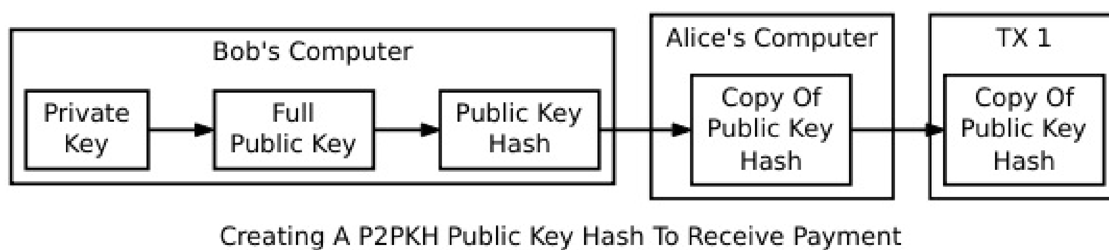


Overview Of Transaction Spending

每一个输出都有一个默认的索引数值，由其在交易中的位置决定，第一个输出就是输出零。输出包含以聪为计数单位的金额，通过满足一个特定条件的 pubkey 脚本来支付。任何可以满足 pubkey 脚本特定条件的人都可以花费对应金额的聪。

输入通过使用交易 id（txid）和输出索引数值（常被称为输出向量的"vout"）来辨识哪一个输出被花费。它同时包含一个签名脚本，这个脚本提供数据作为参数来满足 pubkey 脚本的特定条件。（序列数值和锁定时间是关联的，将会在后续小节中被一起涵盖。）

下图展示了使用这些特性的一个流程，Alice 给你 Bob 发送了一个交易，使得 Bob 之后可以花费这个交易。Alice 和 Bob 使用了支付给公开哈希（P2PKH）这种常用的交易类型。P2PKH 让 Alice 可以发送 satoshis 到一个比特币地址，然后让 Bos 之后可以花费这些 satoshis，其实使用了了一个简单的密码密钥对。



Creating A P2PKH Public Key Hash To Receive Payment

Bob 必须在 Alice 能够发给他交易前就生成一对私有/公开密钥对。比特币使用椭圆曲线数字签名算法（ECDSA），选用的是 secp256k1 曲线，secp256k1 的私钥是 256 位的随机数据。这份数据的副本可以被准确的转化为 secp256k1 的公钥。因为转化可以在之后被准确的进行，公钥是没有存储的必要性的。

然后公钥（pubkey）进行密码哈希。这个公钥哈希之后也可以准确的算出，所以它也是不需要被存储的。这个哈希缩短并混淆了公钥，使得手动的交易更容易并提供了安全性，用以抵抗一些未曾预料的问题，比如在未来可能出现通过公钥重新构建出私钥。

Bob 提供给 Alice 公钥哈希。公钥哈希通过编码后就得到了比特币地址，使用 base58 编码得到的字符串包含了一个地址版本数字、哈希值和一个错误校验的校验和用来捕获错别字。比特币地址可以通过任何媒介传播，包括单向媒介从而省去了消费者与接收者间的沟通，它可以之后编码为任何其他格式，比如一个包含了 `bitcoin:` URI 的二维码。

P2PKH 脚本验证 | P2PKH Script Validation

Developer Guide - Bitcoin

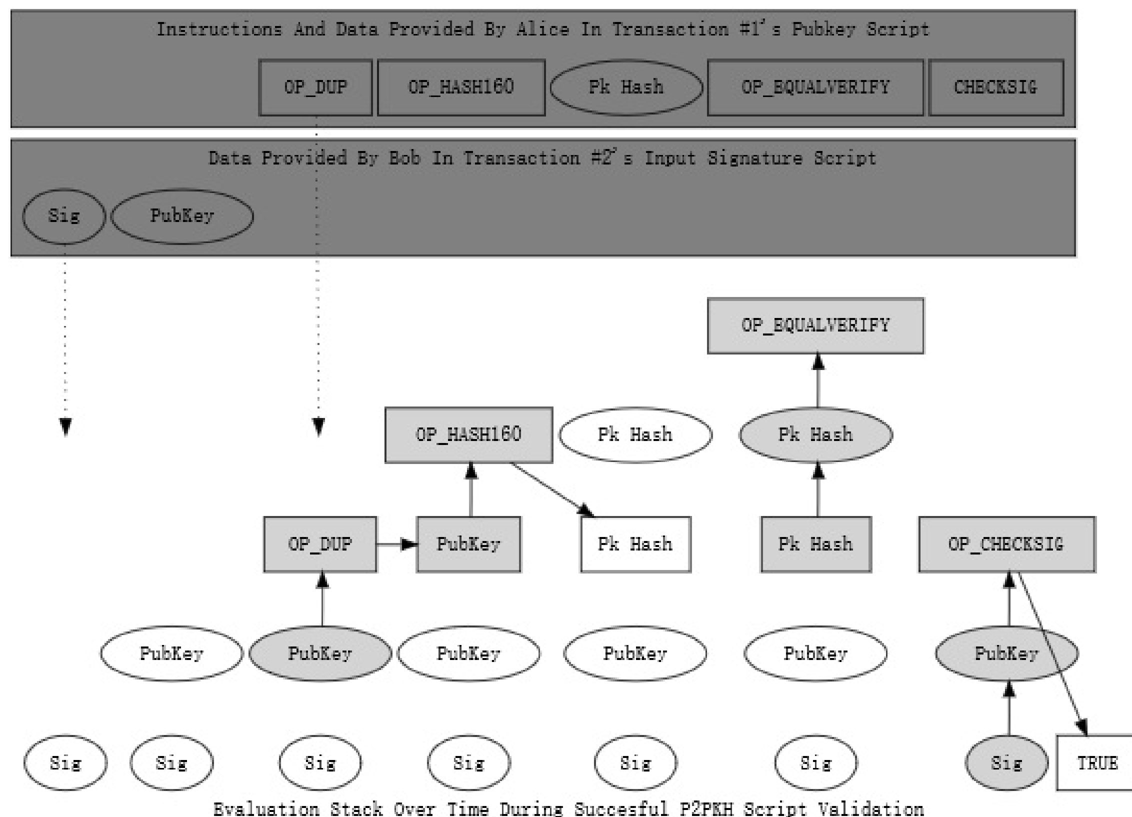
验证过程需要计算签名脚本和公钥脚本，对于一个P2PKH的交易，支出地址的公钥脚本是如下格式

```
OP_DUP OP_HASH160 <PubkeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

交易的花费者的签名脚本会被验证，并作为脚本开头的前缀。 一个P2PKH 交易的签名脚本包括一个secp256k1加密算法的签名和完整的公钥，拼接成以下格式：

```
<Sig> <PubKey> OP_DUP OP_HASH160 <PubkeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

脚本语言是一种类 Forth 栈语言。被设计拥有成无状态和非图灵完备的性质。无状态性保证了一旦一个交易被区块打包，这个交易就是可用的。图灵非完备性（具体来说，缺少循环和 goto 语句）使得比特币的脚本语言更加不灵活和更可预测，从而大大简化了安全模型。为了检测交易是不是有效的，从Bob的签名脚本一直到Alice 的公钥脚本依次执行。下面的图示展示了计算一个标准P2PKH 的公钥脚本。



Evaluation Stack Over Time During Succesful P2PKH Script Validation

签名（来自Bob 的签名脚本）被压入空栈，以为签名只是数据，不需要做任何计算，只是把签名入栈。之后公钥（同样来着签名脚本）被压入堆栈。

- 从Alice 的公钥脚本，开始执行OP_DUP，OP_DUP 被压入堆栈，拷贝一份堆栈顶的数据，在这个例子中，指的是拷贝Bob提供 的公钥。
- 接下来执行的运算是 OP_ HASH160, 把堆栈头的数据进行HASH 运算，在这个例子中，对Bob 提供的公钥进行了HASH运算。
- 接下来Alice 的公钥脚本把Bob在第一笔交易中给他的公钥Hash 压入栈。这时候，已经有了两份Bob 的公钥Hash放置在堆栈的头部。
- 有趣的部分：Alice 的公钥脚本执行 OP_EQUALVERIFY. OP_EQUALVERIFY 等价于执行OP_VERIFY(未展示) 后执行 OP_EQUAL.

OP_EQUAL (未展示)计算堆栈头部的前两个值，在这个例子中，检测了 从Bob 提供的全 公钥生成的公钥Hash 是否和Alice提供的在第一次交易中生成的公钥hash一致。OP_EQUAL 出栈头部的两个值，压入计算结果：0 （否）和 1（真）。

OP_VERIFY(未展示) 检查了堆栈头的值，如果堆栈头为FALSE， 立刻终止全部计算，交易验证失败。否则，以TRUE 值出栈。

- 最后，Alice’s 的公钥脚本执行 OP_CHECKSIG, 该操作检测Bob 提供的签名是不是授权了Bob提供的现在认证的公钥。如果签名和公钥匹配，并且使用所有需要签名的数据生成，则OP_CHECKSIG将值true推送到堆栈的顶部。

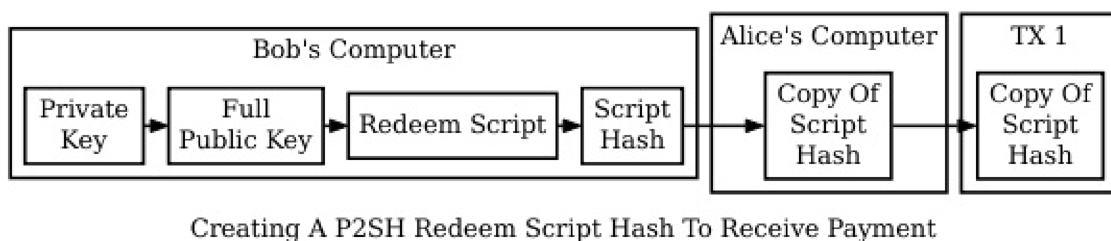
如果经过公钥脚本计算，在堆栈顶部不是False 值，那么验证脚本为有效的（证明交易没有其他问题）。

P2SH 脚本 | P2SH Scripts

公钥脚本是由花费者创造的，花费者对脚本做了什么没有太多兴趣。但是收入者会更关心脚本的情况，如果他们有意愿，他们可以支付者使用一个特定的公钥脚本。不幸的是，自定义的公钥脚本不如一个比特币短地址相方便。而且，在BIP70支付协议被广泛讨论和实行之前，没有标准的方式来在各程序之间传递公钥脚本。

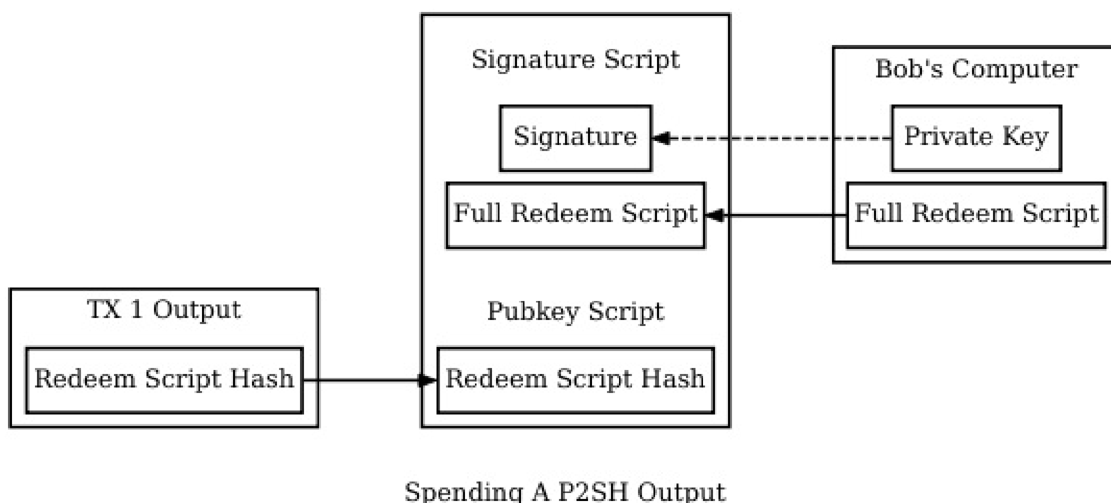
为了解决这个问题，pay-to-script-hash （P2SH）， 支付到脚本哈希 交易在2012 被发明。P2SH使得花费者可以创建一个包含第二个脚本（兑换脚本）的hash 的公钥脚本。

下图展示了基本的几乎和P2PKH一样的，P2SH的工作流程。Bob 创建了一个附带任意Bob想要的脚本的兑换脚本，hash 兑换脚本，然后把兑换脚本的hash 值提供Alice。Alice 创造一个包含Bob的兑换脚本hash 的P2SH类型的支出。



Creating A P2SH Redeem Script Hash To Receive Payment

当Bob想要花掉这笔支出，他在签名脚本中 提供自己的签名和完整的（序列化）的兑换脚本。点对点网络确保了全兑换脚本的hash 值和Alice 放入她的input 的是相同的。如果有初始公钥，像处理兑换脚本一样执行：兑换脚本不返回false，则让Bob花费输出。



Spending A P2SH Output

兑换脚本的hash 和公钥哈希有相同的性质，所以只需要一个用来区别标准地址的小改动，兑换脚本就可以转换成标准比特币地址格式。这使得获取P2SH 样式的地址和获取 P2PKH样式的一样简单。哈希运算还会对兑换脚本中的任何公钥进行模糊处理，因此P2SH脚本获得了和P2PKH 的公钥hash一样的安全性。

标准交易 | Standard Transactions

在早期的几个比特币版本中发现几个严重的漏洞后，加入了一个检测的操作。只有当他们的公钥脚本和签名脚本与一组被认为是安全的模板匹配时，并且其余交易没有违反执行良好网络行为 的另一组规则，才接受来自网络的交易。这个操作名为 isSantard（）测试，通过该测试的交易称为标准交易。

那些没有通过测试的，被称为非标准交易，他们 可能被未使用比特币 core 开发组的默认设置的比特币节点所接受，如果他们包含在区块内，这些交易应该避免进行IsStandard 检测和被处理。

除了增加了通过免费广播有害交易来攻击比特币的困难度，标准交易检测可以防止用户创造阻碍将来增加新的交易特性的交易。比如说，每个交易包括版本号 - 如果用户开始任意改变版本号，则它将作为用于引入向后不兼容特征的工具变得无用。

支付到公钥哈希(P2PKH)|Pay To Public Key Hash (P2PKH)

P2PKH 是最常见的公钥脚本形式，用来发起向多个或者一个地址的支付。

```
Pubkey script: OP_DUP OP_HASH160 <PubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
Signature script: <sig> <pubkey>
```

支付到脚本公钥|Pay To Script Hash (P2SH)

P2SH用来向一个脚本hash 发起交易。每一个标准公钥脚本可以视为以一个P2SH 兑换脚本，但是，在实践中，直到更多的交易类型被认为是标准类型，只有多重签名公钥脚本是合理的。

```
Pubkey script: OP_HASH160 <Hash160(redeemScript)> OP_EQUAL
Signature script: <sig> [sig] [sig...] <redeemScript>
```

多重签名脚本|Multisig

尽管P2SH 多重签名脚本一般用于多重签名的交易，但是这个基础性的脚本也可以用于这种场景：当一个UTXO被使用之前，需要多重签名验证。

多重签名公钥脚本可以一般称为 m-of-n，至少需要m 个匹配公钥，n提供的公钥总数。m 和n都应当根据需要的数量进行从 `OP_1` 到 `OP_16` 运算。

原始的比特币实现中存在一个错位错误，因此，出于兼容性原因， `OP_CHECKMULTISIG` 实际上在堆栈中消耗了m+1个值，因此签名脚本中的secp256k1签名列表必须以额外的值开头（ `OP_0` ），它将被消耗但不被使用。

这个签名脚本必须对应的公钥脚本中出现的公钥顺序，详细细节参见 `OP_CHECKMULTISIG` 。

```
Pubkey script: <m> <A pubkey> [B pubkey] [C pubkey...] <n> OP_CHECKMULTISIG
Signature script: OP_0 <A sig> [B sig] [C sig...]
```

尽管这不是一个单独的交易类型，但是他使用了P2SH 实现了2-of-3 的多重签名：

```
Pubkey script: OP_HASH160 <Hash160(redeemScript)> OP_EQUAL
Redeem script: <OP_2> <A pubkey> <B pubkey> <C pubkey> <OP_3> OP_CHECKMULTISIG
Signature script: OP_0 <A sig> <C sig> <redeemScript>
```

公钥|Pubkey

公钥输出是一个P2PKH 公钥脚本的简化形式，但是安全性不及P2PKH，大部分新交易已经再不适用这种方式了。

```
Pubkey script: <pubkey> OP_CHECKSIG
Signature script: <sig>
```

Null数据|Null Data

在Bitcion Core 0.9.0 和之后版本，可以把任意数据增加到可验证的不可支出公钥。因此当使用该版本的默认设置来传播和挖掘 的Null数据交易类型，不会被全节点存储在UTXO集上。因为Null数据交易类型在TUXO集中不能被自动修剪，这种交易类型适用于UTXO数据库膨胀的场景。然而，如果可以，更好的选择是把数据储存在交易信息之外。

如果null 数据交易符合一致性共识的其他部分，比如没有大于520 Bytes 的提交 数据，一致性共识允许null data的公钥脚本的大小最多10000bytes.

Bitcoin Core 0.9.x 到0.10.0 版本将会默认设置中转和打包null data 交易类型，支持单次提交数据多达40bytes和一个收入为0聪的交易结果。

```
Pubkey Script: OP_RETURN <0 to 40 bytes of data>
(Null data scripts cannot be spent, so there's no signature script.)
```

Bitcoin Core 0.11.x将默认值增加到80字节，其他规则保持不变。

只要不超过总字节限制，Bitcoin Core 0.12.0 默认支持最多83个字节，任意数量的数据推送的null 数据交易。必须仍然只有一个空数据输出和0聪的交易。

Bitcion Core 的 `-datacarriersize` 配置选项允许设置最大体积来限制Null数据交易的广播和打包。

非标准交易 | Non-Standard Transactions

如果在输出中使用了任意非标准公钥脚本，使用Bitcoin Core默认设置的节点和矿工将不接受，不广播，不挖掘这个交易。当试图如此做时，会收到错误信息。

如果创建了一个兑换脚本，对其hash运算并应用到P2SH输出中，点对点网络将只能看到hash值。所以无论兑换脚本的内容是什么，都会被验证未有效输出。从Bitcoin Core 0.11之后，允许支付到非标准脚本，几乎所有的兑换脚本都可以用来支付。唯一的例外是 使用未分配的NOP操作码的脚本，这些操作码保留用于将来的软分叉，所以只能被不遵守标准 mempool 协议的节点广播和打包。

注意：标准交易类型旨在保护比特币网络，而不是防止用户犯错。事实上，创建 支付到无法花费地址的标准交易是十分容易的。

从Bitcoin Core 0.9.3开始，标准交易也必须满足以下条件：

- 打包好的交易必须是完成状态的。它的锁定时间必须是过去的某个时间（或小于或等于当前块的高度），否则它的所有序列号都必须是0xffffffff。
- 交易必须小于100,000个字节。这是典型的单输入，单输出的P2PKH交易大约200倍。
- 每个交易的签名脚本必须小于1,650字节。这对于允许使用压缩公钥的P2SH类型15of15的多重交易来说，足够大了。
- 需要多于3个公钥的裸（非P2SH）多重交易当前是非标准交易。
- 交易的签名脚本只能将数据推送到脚本计算堆栈。除了仅将数据推送到堆栈的操作码，它不能推送新的操作码。
- 交易不允许这种输出：接收少于它在一个典型的输入中花费的1/3 聪。目前来说，相当于在默认广播费的bitcoin Core 节点中，至少需要 546聪，Null 数据标准是例外情况，因为null 数据交易的输出必须为0。

签名哈希类型 | Signature Hash Types

`OP_CHECKSIG` 从每个计算的签名中提取一个非堆栈参数，这使得签名者可以选择签署交易任意部分。这种设置使得被签名的部分不会被篡改，同时，签名者可以有选择的让其他人修改他们的交易。

各种的签名选项称为哈希前面类型，目前有三种可用：

- `SIGHASH_ALL` ,是默认设置，该签名类型保护了输入和输出，只有签名脚本是可以更改的。
- `SIGHASH_NONE` 签名所有输入，不签名输出，除非其他签名使用其他签名哈希flag保护输出，允许任何人改比特币的去向。
- `SIGHASH_SINGLE` 只签署一个和input 对应的输出（输出有和i输入有相同的输出索引（output index））确保没人可以改变签名人所属的交易部分， 交易的其余部分都是可以更改的。对应的签名输出必须存在或者值为1来绕过比特币的安全策略。使用这种方法签名的input 和其他input 都包括在签名中，但其他input 的序列号不包括在签名中，可以更新。

以上三种基础签名哈希类型可以结合 `SIGHASH_ANYONECANPAY` flag 创造三种新的类型：

- `SIGHASH_ALL|SIGHASH_ANYONECANPAY` 签名所有output 和一个input，并且允许任何人添加和删除其他输入和输出。所以每个人可以贡献自己比特币放入到交易中，但是不能改变已经签名的部分的交易 的流向的数量。
- `SIGHASH_NONE|SIGHASH_ANYONECANPAY` 只签名自己的input，允许其他人修改其他输入和输出，所以，得到这个签名的人，可以任意的花掉这笔input。
- `SIGHASH_SINGLE|SIGHASH_ANYONECANPAY` 签名一个input和对应的output，允许其他人增加或者删除其他input。

因为每个input都可单独签名，所以，当这些签名保护了交易的不同部分时，一个多个input 的交易可以有多个签名类型。例如，一个只有一个input 的交易签名为 `SIGHASH_NONE` ，所以，交易的output可以被矿工修改并被打包。对比来看，一个有两个input 的交易，一个签名为 `SIGHASH_NONE` 另一个签名为 `SIGHASH_ALL` ，all 的签名者可以不经None 的同意，花掉这笔钱，但是其他人不可以。

锁定时间和序列号|Locktime And Sequence Number

所有的签名类型都会签定交易的锁定时间(Bitcoin Core 的源代码里称为 nLockTime.) locktime 代表交易被允许被被打包的最早时间。

Locktime 允许签名者创建一个时间锁定交易。因为只会在将来生效，这给签名者一个的反悔的机会。

如果其中任何一个签名者反悔了，他可以创建一个没有locktime 的交易。因为新创建的交易可以花掉旧交易的那部分input，所以旧交易在lock time解锁后 找不到可以花掉的input，旧交易就失效了。

当心 locktime所 的快失效的情况，点对点网络允许locktime 失效的前两个小时，打包被锁定的交易，此外区块产生的时间间隔也是不确定的。所以如果想取消交易，一定要在锁定时间的几个小时之前进行。

Bitcoin Core 的早期版本提供了一个可以防止签名者使用上述方法取消locktime 交易的功能。出于防止拒绝服务攻击的原因，这个功能被禁用了。该系统还留下了这样的设置，每个输入会分配一个四字节的序列号。序列号的目的旨在允许多个签名者同意更新交易。他们可以将自己的sequence number设置为四字节的的无符号最大值(0xffffffff),使得交易的locktime 仍然有效的情况下，打包交易进块。

即使今天，如果所有的input 的sequence number都是最大值，locktime锁就会失效。所以如果想使用locktime，至少一个input的sequence number要小于最大值。由于sequence number 不用于其他目的，任何sequence number 为零的交易都会启动locktime 功能。

Locktime 本身是一个无符号的四字节整数，可以通过以下两种方式解析：

- 如果locktime 小于5亿，locktime 被认为是区块的高度。交易只能被大于等于此高度的区块打包。
- 如果locktime 大于等于5亿，locktime 为使用Unix 时间格式（从1970-01-01 00:00 UTC 经过的秒数，目前超过12.95亿）解析。交易可以添加到任何大于此时间的区块中。

交易手续费及找零 | Transaction Fees And Change

交易的手续费是基于交易的体积计算的。每字节的手续费是根据当前开采区块对空间的需求计算的，费用随着需求增加而增加。何在去快链部分解释的一样，交易费支付给打包出区块的矿工，最终每个矿工选择接受他们将接受的最低交易费。

还有一个概念是 “high priority transactions”，高优先级交易，指的是包含那些很久未交易的比特币的交易。

过去这些交易都是免费的，在Bitcoin Core 0.12 之前，每个块的50KB用来优先打包这些交易。但是现在默认情况下这个值为0K，在优先区域打包满之后，所有交易根据他们的每字节费用进行排序，先打包交易费高的交易，直到所有可用空间已经填满。

从比特币0.9开始，一笔交易需要的最低费用（目前是1000 聪）会被网络广播，任何只支付最低费用的交易会等待很长时间，直到有一个块有足够空间来打包它。请参阅付款验证部分来了解最低交易费的重要性。

因为每个交易花掉了Unspent Transaction Outputs 未花费输出集合(UTXOs)中的比特币，并且应为UTXO只能被花费一次，每次交易UTXO 的一部分会给矿工作为交易费。很少有人想要支付的金额和自己的UTXO 是匹配的，所以大部分的交易会有一个用来找零的ouptut。

找零output和其他output 一样，只是将交易剩余的比特币返还给支付者。他们可以使用和在input是在UTXO中相同的P2PKH 公钥哈希和P2SH 脚本哈希，但是出于下一节要阐述的原因，强烈不建议这样做。

避免 公钥和私钥重用 | Avoiding Key Reuse

交易中，交易的支付者和接收者会暴露自己使用 的公钥或者钱包地址。这使得任何一个人可以使用公开的区块链信息来跟踪一个人的过去和将来的相关交易。

如果多次重用相同公钥 ，或者使用比特币的钱包地址（公钥的哈希值）作为静态支付地址，其他人可以轻易的跟踪钱包所有者的消费习惯，拥有的比特币数量。

其实我们也可也避免陷入这种情况。如果每个公钥地址只使用两次，一次接受付款，一次花掉比特币，用户的金融隐私会得到有效保护。

或者更进一步，在接受付款或者创建找零钱包时，使用新的公钥或者唯一的钱包地址并且结合稍后讨论的其他技术（CoinJoin 或者merge avoidance）结合起来。这使得使用区块链信息来有效的跟踪某个用户的金融往来变得极其困难。

除了隐私性，避免重用公钥还有安全性的原因。防止根据公钥或者签名的对比从而构建私钥进行攻击（下面的这两种总结的更一般的假想攻击的情景）。

1. 唯一（非重用）的P2PKH 和P2SH 地址保护了的第一种攻击。通过隐藏ECDSA 公钥直到第一次花掉钱包里的比特币。除非可以在一到两个小时，一个交易被区块链有效的保护需要的时间，重建私钥，否则攻击是十分无用且低效的。
2. 唯一（非重用）私钥保护了第二种攻击。每个私钥仅生成一个签名，所以攻击者不会得到私钥的后续签名，来进行基于对比（私钥和签名）的攻击。现存的基于比较（私钥和签名）的攻击只在这种情况是可行的：当签名中使用了不充分的熵，或者通过某种手段，比如旁道攻击。，暴露了使用的熵。

处于安全性和金融隐私的原因，强烈建议开发应用时，避免重用公钥。并且尽可能的阻止用户使用重用地址。如果您的应用程序需要提供一个固定的付款URI，请参阅下面的 `bitcoin:` URI section 部分。

交易延展性 | Transaction Malleability

没有一个比他被的签名哈希脚本可以保护签名脚本，所以给有限的DoS攻击留下了方便之门，也称之为交易的延展性。签名脚本（signature script） 包含一个secp256k1的椭圆曲线加密签名，但是不能签名签名脚本自己，这使得攻击者可以对交易进行非功能的修改但交易依旧有效。例如，攻击者可以向签名脚本添加一些数据，这些数据会在之前的公钥脚本计算前被删除。

尽管修改是没有影响交易功能，所以攻击者既不能修改input 也不能修改output，但是他们修改了交易的hash值。因为交易会通过交易hash链接之前的交易作为交易标识符（txid），所以一个被修改的交易不具有创建者期望的交易id（txids）。

这对大部分交易都是不成问题的，因为交易应该被迅速的打包到区块链中。但是如果output在交易被打包之前就被花出去了，这就会是一个严重的问题。

比特币的开发者一直努力降低标准交易类型的延展性，但是完整的修复交易的延展性仍然处于规划阶段。目前，新的交易不应当依赖尚未打包到区块的交易，尤其是处于高风险的大额比特币交易。

交易的延展性也影响交易的追踪，Bitcoin Core 客户端的的RPC接允许根据交易id （txid）查找交易。但是如果txid因为交易被修改而变更，客户端会认为交易从网络中消失了。

目前跟踪交易的最佳实践应当是跟踪作为交易input的未花费输出（UTXOS），因为utxo 只能在有效交易的情况下被改变。

最佳的实践方案进一步规定，如果一个交易看上去从网络中消失并且需要重新广播，那么它将使丢掉的交易无效的方式重新发布。一种总是可用的方式是，重复广播丢失的交易，保证相同的input 和 output。

合约 | Contracts

合约（contract）是去中心化的比特币系统保证有效力的财务交易。比特币合约通常可以设计为对外部代理（比如法院）最小依赖。这大大降低了交易中对于未知交易实体的交易风险。

下面的小节将描述已经使用的各种的比特币合约。因为除了交易本身，合约还涉及现实中的人，我们设计成以下的故事模板。

除了下面描述的合约类型，还提出了许多其他合约类型。 其中几个收集在比特币维基的合约页面。

托管和仲裁 |Escrow And Arbitration

顾客Charlie 想要从老板Bob买商品，但是他们之间相互猜疑。所以他们使用合约来保证Charlie 能拿到商品，Bob 能拿到钱。

一个简单的合约可以这样设定，Charlie支付比特币到一个多重签名地址，只有Bob 和Charlie都签名时output 才可以被花费。也就是说，除非Charile拿到商品后Bob 才会被支付，还可能是Charlie没有拿到商品，但是支付的钱也拿不回来了。

这个简单的合约没有解决问题，所以他们找到了Alice作为仲裁者来创建托管交易。Charlie 支付比特币到一个2-of-3 d 多重签名脚本，（只有两个及以上签名时，output才可以花费）。如果没有问题，Charlie 可以签名支付给Bob，商品有问题时，Bob 退回Charlie 的钱。如果有分歧，Alice 可以决定谁得到支付的比特币。为了创造一个多重签名地址（output），三个交出他们的公钥，Bob 创建如下的P2SH的赎回脚本：

```
OP_2 [A's pubkey] [B's pubkey] [C's pubkey] OP_3 OP_CHECKMULTISIG
```

（把公钥的推入验证栈的操作码（Opcode）没有展示）

`OP_2` 和 `OP_3` 把真实的数字2 和3 放入栈内。OP_2 表明需要两个匹配的签名才能生效， `OP_3` 表明需要提供三个公钥（未经过哈希运算的）。这就是2 -of - 3 的多重签名脚本，更一般的叫法是 m-of -n 公钥脚本（m是需要的最小匹配的签名数量，n是提供的公钥数量）

Bob 把这个赎回脚本给Charlie，确保他的公钥和Alice 的公钥包含在脚本内。之后，Charlie hash运算这个赎回脚本，创建了P2SH 交易的赎回脚本，然后为这笔交易支付比特币。Bob 看到P2SH交易被区块链打包确认后，交付商品。

不幸的是，运送过程中商品有了轻微的损耗，Charlie 希望要全部退款，Bob 认为退还10% 才是合理的。于是他们找到Alice 仲裁这笔交易。Alice 向Charlie 所要之前由Bob 生成并由Charlie 检查确认过的赎回脚本和商品损耗的证据。

Alice 经过查看证据，认为40%是一个合理的数值。所以 Alice创建并签名了一个有两个output的交易，支付60%的比特币到Bob 的公钥地址，支付剩余部分到 Charlie 的公钥地址。

在签名脚本中，Alice 写入她自己的签名和Bob之前创建的未哈希的序列化的赎回脚本的副本，Alice 把这个不完整交易的副本分别给Charlie 和Bob。Charlie 和Bob中的任意一个人都可以通过自己签名的方式，创建下面的签名脚本:

```
OP_0 [A's signature] [B's or C's signature] [serialized redeem script]
```

(把签名和赎回脚本压入验证栈的操作码没有展示，OP_0 是为了解决原始实现中的一个错误，由于兼容性原因，后期版本不得不保留i这个操作符，请注意，签名脚本内的签名的顺序和赎回脚本中的公钥顺序比喻一致，有关详细内容请参阅 `OP_CHECKMULTISIG`)

当交易广播到网络时，每个节点可以根据Charlie的前序交易P2SH脚本来检查签名，确保赎回脚本和前序交易的赎回脚本的hash匹配。然后使用两个签名作为输入数据，计算赎回脚本。如果赎回脚本是有效的，交易中两个output会作为可支付余额出现在Bob 和Charlie的钱包中。

如果Alice 反悔，创建并且签署了一个Bob 和Charlie 都不同意的交易，比如Alice 自己私吞了这笔钱。Bob 和Charlie 可以找到一个新的仲裁者，使用新的仲裁者公钥，制作新的多重签名公钥。这意味着，Bob 和Charlie 不需要担心仲裁者一个人可以偷钱。

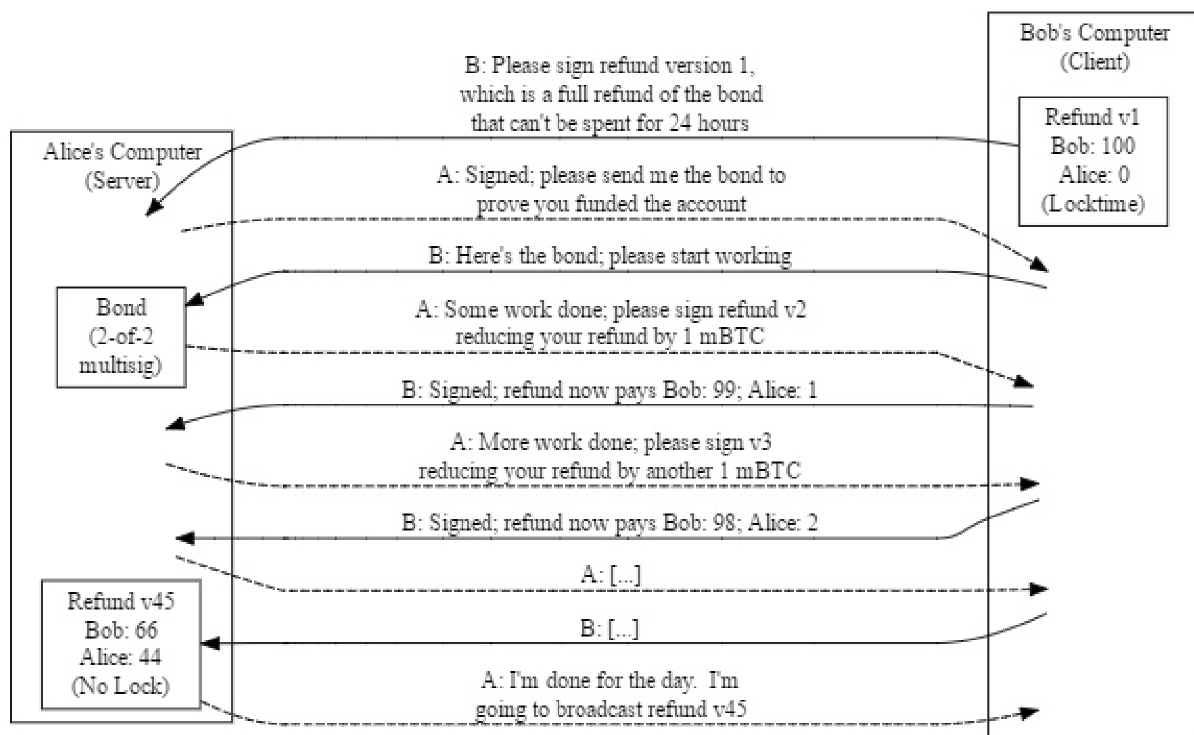
微支付通道 | Micropayment Channel

Alice 为Bob 工作，兼职管理论坛。每当有人在Bob的论坛发帖，Alice会检查帖子是不是攻击性或者垃圾内容。但是Bob 经常忘记支付工资，所以Alice 要求每当她批准或者删除摸个帖子时立即支付工资。Bob 由于交易费拒绝了，因为数百笔小额付款会产生上万的交易费。所以Alice 建议使用微支付通道技术。

Bob向Alice 索要公钥，然后创建如下交易。

bond transaction： 通过P2PH支付100 聪到一个2 -of -2 的公钥脚本，需要Alice 和bob 的共同签名。 广播这个交易会使得 100 聪成为Alice 的"人质"，所以这时Bob 不会广播这个交易。

refund transaction： 第二个交易在锁定时间一天后，支付全部bond transaction的output（减去交易费）返还给Bob。Bob 不能自己签署退款交易，所以她要让Alice签名。如下图所示



Bitcoin Micropayment Channels (As Implemented In Bitcoinj)

Alice 检查refund 交易是在24小时后，签署后交还给Bob交易副本。然后Alice 向Bob 索要Bond transaction 查看refund 是否能花掉Bond transaction 的钱。现在Alice 可以向网络广播这个交易，确保Bob 的100聪仔交易锁定到期之前不会被花到其他地方。对于Bob，除了一部分交易费，没有花掉任何比特币，他能够在锁定时间之后（24小时）广播refund交易来获得全额退款。

现在Alice完成了价值1 聪的工作，Alice 要求Bob 重写并签名一份新的refund 交易。版本2交易 分配1聪给Alice。99 聪给Bob。这个交易没有锁定时间，所以Alice 可以在任何时间签名后广播到网络中（但是Alice 没有立即这样做。）

Alice 和Bob 重复进行 "工作-支付" 的步骤。直到这一天结束或者锁定时间快失效时，Alice 签署并且广播最后一个版本的refund，Alice 和Bob 分配到了最后一份refund 分配的比特币。第二天，Alice 开始新的工作，他们创建了一个新的 微支付通道合约。

如果在第一版refund 的锁定时间24小时内，Alice没有成功广播任何一个版本的refund 交易，Bob 可以在锁定时间失效后广播第一版的refund拿到全部退款。这也微支付通道为什么只适合小额交易的原因，如果Alice 的网络有问题，没能在锁定失效前广播有利自己的refund，Alice 就会遭受损失。

正如之前所讨论的交易的延展性问题，微支付通道因此也受到了限制。如果有人利用交易延展性攻击fund 和refund 的联系，Alice 可以在没有进行相应工作的情况下，成功劫持Bob 的100 聪。

此外，对于大额的支付，比特币的交易费相对交易金额是极低比例，更合理的方式是分别立即各个广播交易。

Resource:

java 库 bitcoinj 提供了一整套微支付功能，下面是（基于Apache 许可的）一个微支付通道的实现和教程：a tutorial

爱的色放

一个比特币钱包可以是指一个钱包程序或者一个钱包文件。钱包程序创建公钥来接收比特币，并且使用对应的私钥来花费它。钱包文件为钱包程序存储私钥，也可能会存储交易相关的一些信息。

钱包程序和钱包文件在下面被分为不同的章节，本文章期望总是可以分的很清楚，无论我们是谈论钱包程序或者钱包文件。

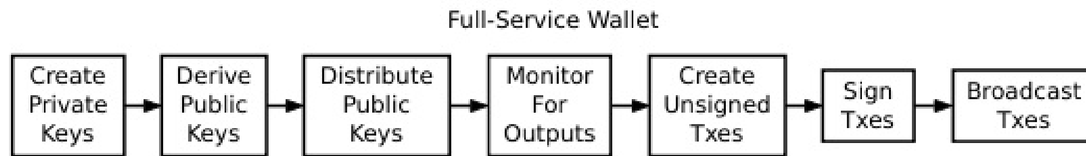
许可接收和花费比特币是钱包软件的唯一基础特性，但是一个钱包程序不一定非要兼具这两部分。可以两个钱包程序一同运作，一个程序来发布公钥用于接收比特币，而另一个程序对交易做签名来花费比特币。

比特币钱包同时需要与点对点网络做交互，从而通过区块链获取信息以及广播新的交易。然而，用于发布公钥的程序及签名交易的程序其自身并不需要与点对点网络做交互。

这让我们有三个必要但彼此分离的钱包组件：一个公钥发布程序，一个签名程序，一个网络程序。接下来的小节内，我们将讨论这些组件的常见组合。

注：我们谈论发布公钥是作为一种通称。在大多数情况下，P2PKH 或者 P2PH 的哈稀值会被发布出去而非公钥，真正的公钥只有在对应地址管理的比特币被花费时才真的被发布。

包含以上谈到的三部分的钱包是一个最简单的钱包：它生成私钥，生成对应的公钥，按照需要对公钥进行发布，监听以这些公钥作为输出的交易，创建交易并对其进行签名，广播已签名的交易。



截止本文档撰写，几乎所有的钱包都可以用作完整服务的钱包。

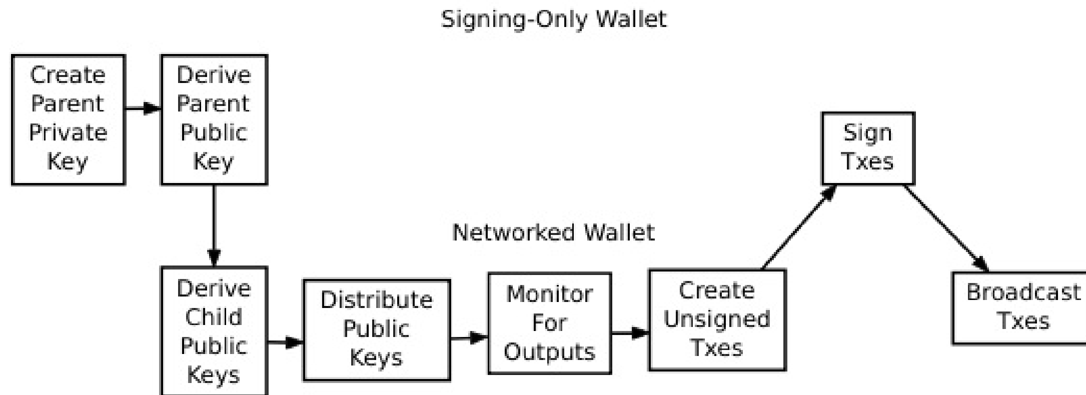
全服务钱包的主要好处是它们很便于使用。一个单一的程序满足了用户接收和花费比特币的所有需求。

全服务钱包的主要坏处是它们将一台存储私钥的设备连接到了网络。这些设备上的存在弱点是很常见的，而且网络连接使得从存在弱点的设备传输私钥到攻击者更为容易。

为了阻止偷窃，很多钱包程序提供给用户对包含密钥的钱包文件进行加密的选项。当钱包没有在使用时，这种保护是有效的，但是它无法对抗专门截获加密密码或者从内存中读取已解密私钥的攻击。

为了提升安全性，私钥可以在更为安全的环境中由有单独的程序生成并做存储。只用于签名的钱包与一个可以进行点对点网络交互的钱包协同工作。

只用于签名的钱包通常使用确定性密钥创建方式（在稍后的小节中会讨论）来创建父私钥和公钥，之后还可以再创建子私钥和公钥。



当第一次运行时，只签名的钱包创建一个父私钥并将其对应的公钥传输给联网的钱包。

网络钱包使用父公钥生成子公钥们，可以作为可选操作将其发布，监听这些公钥的输出，创建花费比特币的未签名交易，然后将未签名的交易传输给签名的钱包。

通常，用户有机会使用签名钱包对未签名交易的细节（特别是输出细节）做审核。

在可选的审核步骤之后，签名钱包使用父私钥生成相关的子私钥来对交易做签名，再将已签名的交易送回给联网钱包。

然后联网钱包将已签名交易广播至点对点网络。

接下来的小节将讨论两种常用的签名钱包的变体：离线钱包和硬件钱包。

一些完整服务的钱包也是按照两个分离的钱包程序来运行的：一个程序扮演只用于签名的钱包（通常被叫做“离线钱包”），另一个程序扮演网络钱包（通称被叫做“在线钱包”或“监听钱包”）。

离线钱包的得名因为它是运行在一台不联网的设备上，极大程度的抵御了攻击的媒介。按照这样的方式，通常需要用户通过可移除的媒介，比如 USB 驱动，来控制所有数据的传输。用户操作的流程大致是这样的：

1. （离线）在一台设备上禁用所有的网络并安装钱包软件。在离线模式下开启钱包软件，用来创建父私钥和公钥。复制父公钥到可移除的媒介。
2. （在线）在另一台设备安装钱包软件，这一台设备是联网的，然后从可移除的媒介导入父公钥。如同一个完整钱包一样，发布公钥来接收付款。当准备花费接收到的比特币时，通过钱包填写输出的细节然后生成未签名的交易，将其保存至可移除媒介。
3. （离线）在离线这端打开未签名的交易，审核交易的输出细节确保数目和地址都是正确的。这阻止了在线钱包中的恶意软件欺骗用户将交易信息发送给攻击者。审核过后，签名交易并将其保存至可移除媒介。 4.（在线）在在线钱包中打开交易，以便其在点对点网络中广播。

离线钱包的主要好处是它比完整一个钱包提供了安全性。因为离线钱包未被危害到（或破坏），用户在签名前都审核所有的交易信息，那么用户的比特币就是安全的，就算在线钱包已经被危害到。

离线钱包主要的弊端就是它很麻烦。为了最大的安全性，它需要用户为了离线的事务贡献一台电脑。为了花费比特币，离线设备必须开启，用户必须在离线设备与在线设备间手动的拷贝数据。

硬件钱包是实用单独的硬件来运行的只用于签名的钱包。因为其只运行钱包使得它可以消除很多操作系统为了日常使用而设计时存在的潜在危害，这允许它可以安全直接的与其它设备进行沟通而不需要用户手动的传输数据。用户的流程大致如下：

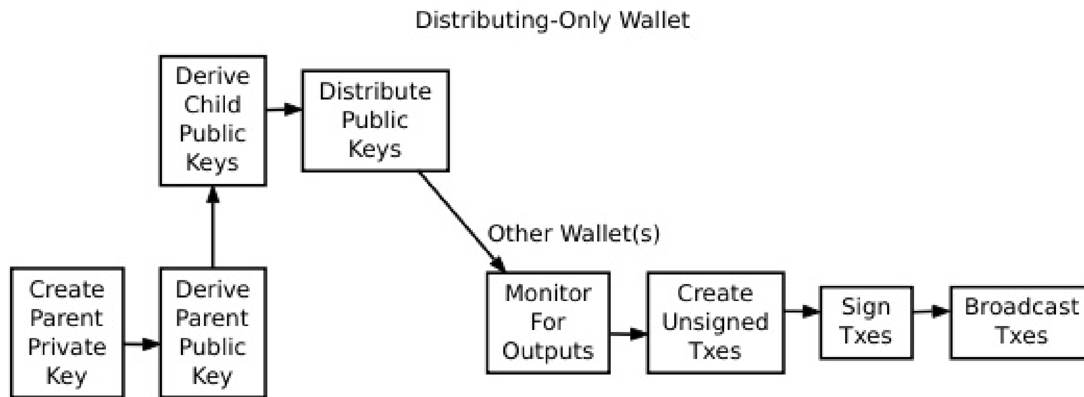
1. （硬件钱包）创建父私钥及公钥。连接硬件钱包到联网设备使之可以获取到公钥。
2. （联网设备）如同一个完整的钱包一样，发布公钥来接收付款。当准备花费接收到的比特币时，填写交易的细节，连接硬件钱包，点击发送。在线钱包会自动的发送交易信息给硬件钱包。
3. （硬件钱包）在硬件钱包的屏幕上审核交易信息。一些硬件钱包会要求密码或者 PIN 码。硬件钱包签名交易并将其传输给在线钱包。
4. （联网设备）在线钱包接收到来自硬件钱包的已签名交易，并将其广播至网络。

硬件钱包的主要好处是它在保证了如同离线钱包的安全性同时还没有离线钱包那么麻烦。

硬件钱包的主要弊端还是其用起来会有些麻烦。即便它已经比离线钱包好很多，但是用户仍然要购买一个硬件设备，无论如何都要携带着它以便支付时使用。

另一个弊端（期望只是暂时的）就是只有非常少的钱包支持了硬件钱包，尽管大部分钱包程序生成至少会支持一种硬件钱包。

在类似 webserver 这种无法保证安全的环境中运行的钱包程序，可以被设计为只发布公钥（包括 P2PKH 或者 P2SH 地址），然后其他什么都不做。有两种常用的方式来设计这种极简的钱包：



- 提前生成一个包含一定数目公钥或地址的数据库，然后使用数据库内的一条记录发布一个请求脚本或地址。为了避免重复使用密钥，webserver 应该持续跟踪已使用的密钥并确保不会使公钥用完。这会比第二种方式中使用父公钥的方式简单不少。
- 使用一个父公钥来创建子公钥。为了避免密钥的重复，需要一个方法来确保一个公钥不会被发布两次。这可以是数据库中每个已发布密钥的记录或者一个递增的数字来作为密钥的索引值。

任何一个方法都产生了不少值得注意的问题，尤其是数据库被用于关联每笔单独公钥的进账跟踪时。查看支付流程章节获得更多细节。